Object Oriented Programming (Python)

It is a programming paradigm - A way to write programs

Class is a template and Object is **Instance** of that class.

We can have multiple instances of a class

In [2]:
```
class Student:  # Class named Student
    pass
```

In [3]:
```
s1 = Student() # s1 is instance of "Student" class.
```

In [7]:
```
print(type(s1))
```

```
<class '__main__.Student'>
```

s1 belongs to the class *Student*

In [8]:
```
l = list()
print(type(l))
```

```
<class 'list'>
```

l belongs to the class list()

Infact everything in python is a class

Attributes in Class

In [4]:
```
s1.name = "Rajat"
```

In [5]:
```
s1.__dict__
```

Out[5]:
```
{'name': 'Rajat'}
```

In [ ]:
```
s2 = Student()
s2.name = 'Rajat'
s2.rollno = '123'
```

name and rollno are attributes(data) of the object s2 of class Student

In [7]:
```
s2.__dict__
```

Out[7]:
```
{'name': 'Rajat', 'rollno': '123'}
```

Those attributes are stored as key-value pairs as dict.

object.key = value

In [8]:
```
hasattr(s1, 'name') # check if s1 have the attribute named "name".
```

Out[8]: True

In [15]:
```
hasattr(s1, 'rollno')
```

Out[15]: False

In [9]:
```
getattr(s1, 'name') # Alternate of object.key
```

Out[9]: 'Rajat'

In [17]:
```
getattr(s2, 'rollno')
```

Out[17]: '123'

In [18]:
```
getattr(s1, 'rollno')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-03a6c151f88f> in <module>
----> 1 getattr(s1, 'rollno')

AttributeError: 'Student' object has no attribute 'rollno'
```

In [19]:
```
# dont want an error
getattr(s1, 'rollno', 'not found')
```

Out[19]: 'not found'

In [25]:
```
# delete attribute
delattr(s1, 'name')
```

In [26]:
```
s1.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-bfa1764dcbdc> in <module>
----> 1 s1.name

AttributeError: 'Student' object has no attribute 'name'
```

In [27]:
```
class student:
    name = 'Rajat Bansal'
    pp = 80

    def isPassed(self):
        self.percentage = 90
        if self.percentage > self.pp:
            return True
```

Class and Instance Methods

In [49]:
```python
class Student:
    passingper = 40
    def StudentDetails(self):  # Instance Method
        self.name = 'Rajat Bansal'
        print(self.name)

        self.percentage = 96
        print(self.percentage)

        pass

    def isPassed(self):  # Instance Method
        if self.percentage >= self.passingper: # or can use Student.passingper
            print("passed")
        else:
            print("Not Passed")
```

In [50]:
```python
s1 = Student()
s1.StudentDetails()
```

```
Rajat Bansal
96
```

In [51]:
```python
s1.isPassed()
```

```
passed
```

Instance Methods are those which do require the properties of object and we need to pass the object itself to them.

Static Methos

These methods are which which do not require properties from the object itself so we do not need to pass 'self' to them. But this can be done by **@staticmethod** decorator. this decorator ignores the first argument from the method.

In [74]:
```python
class Student:
    passingper = 40
    name = 'my name'
    def StudentDetails(self):  # Instance Method
        self.name = 'Rajat Bansal'
        print(self.name)

        self.percentage = 96
        print(self.percentage)

        pass

    def isPassed(self):  # Instance Method
        if self.percentage >= self.passingper: # or can use Student.passingper
            print("passed")
        else:
            print("Not Passed")

    ## Static Method
    @staticmethod  # this ignores the first argument in the function
    def welcometoschool():
        print('Welcome to School')
```

In [69]:
```
s1 = Student()
```

In [67]:
```
s1.welcometoschool()
```

```
Welcome to School
```

In [70]:
```
s1.name
```

Out[70]:
```
'my name'
```

In [72]:
```
s1.StudentDetails() # name updated by the method
```

```
Rajat Bansal
96
```

In [73]:
```
s1.name
```

Out[73]:
```
'Rajat Bansal'
```

"init method"

In [75]:
```
class Student:

    def __init__(self, name, rollno):
        self.name = name
        self.rollno = rollno

    passingper = 40
    name = 'my name'
    def StudentDetails(self):  # Instance Method
        self.name = 'Rajat Bansal'
        print(self.name)

        self.percentage = 96
        print(self.percentage)

        pass

    def isPassed(self):  # Instance Method
        if self.percentage >= self.passingper: # or can use Student.passingper
            print("passed")
        else:
            print("Not Passed")

    ## Static Method
    @staticmethod  # this ignores the first argument in the function
    def welcometoschool():
        print('Welcome to School')
```

In [77]:
```
s1 = Student('Rajat', 123)
```

Classmethods

Classmethods manipulates the input and return an object of any class.

In [90]:
```python
from datetime import date
class Student:

    def __init__(self, name, age, rollno):
        self.name = name
        self.age = age
        self.rollno = rollno

    # classmethod
    @classmethod
    def fromBithYear(cls, name, year, rollno):
        return cls(name, date.today().year - year, rollno)



    def StudentDetails(self):  # Instance Method
        self.name = 'Rajat Bansal'
        print(self.name)
        print("age", self.age)
        self.percentage = 96
        print(self.percentage)

        pass

    def isPassed(self):  # Instance Method
        if self.percentage >= self.passingper: # or can use Student.passingper
            print("passed")
        else:
            print("Not Passed")

    ## Static Method
    @staticmethod  # this ignores the first argument in the function
    def welcometoschool():
        print('Welcome to School')
```

In [82]:
```python
s1 = Student.fromBithYear('Rajat', 2001, 25)
```

In [83]:
```python
s1.StudentDetails()
```

```
Rajat Bansal
age 20
96
```

Public and Private Modifiers

In [28]:
```python
from datetime import date
class Student:
    __passingpercentage = 40 # private Variable

    def __init__(self, name,age, rollno):
        self.__name = name
        self.age = age
        self.rollno = rollno

    # classmethod
    @classmethod
    def fromBithYear(cls, name, year, rollno):
        return cls(name, date.today().year - year, rollno)



    def StudentDetails(self):  # Instance Method
        print(self.__name)
        print("age", self.age)
        self.percentage = 96
        print(self.percentage)

        pass

    def isPassed(self):  # Instance Method
        if self.percentage >= self.passingper: # or can use Student.passingper
            print("passed")
        else:
            print("Not Passed")

    ## Static Method
    @staticmethod  # this ignores the first argument in the function
    def welcometoschool():
        print('Welcome to School')
```

In [29]:
```python
s1 = Student('Rajat', 20, 251)
```

In [30]:
```python
s1.StudentDetails()
```

```
Rajat
age 20
96
```

In [31]:
```python
s1.name = "Rohan"
```

In [32]:
```python
s1.name
```

Out[32]:
```
'Rohan'
```

In [33]:
```python
s1.__name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-33-04dad977536f> in <module>
----> 1 s1.__name

AttributeError: 'Student' object has no attribute '__name'
```

In [25]:
```
# access outside
# called Name Bangling Syntax: object._className__varName

s1._Student__name
```

Out[25]:
```
'Rajat'
```

In [72]:
```
class Fraction:

    def __init__(self, num = 0, den = 1):
        if den == 0:
            # Thorow error
            den = 1
        self.num = num
        self.den = den

    def print(self):
        if self.num == 0:
            print(0)
        elif self.den == 1:
            print(self.num)
        else:
            print(self.num, "/" , self.den)
    def simplyfy(self):
        if self.num == 0:
            self.den = 1
            return
        curr = min(self.num, self.den)
        while curr > 1:
            if self.num % curr == 0 and self.den % curr == 0:
                break
            curr -= 1
        self.num = self.num // curr
        self.den = self.den // curr

    def add(self, other):
        newNum = other.den * self.num + other.num * self.den
        newDen = other.den * self.den

        self.num = newNum
        self.den = newDen

        self.simplyfy()

    def multiply(self, other):
        self.num = other.num * self.num
        self.den = other.den * self.den

        self.simplyfy
```

In [77]:
```
f1 = Fraction(2,3)
f2 = Fraction(1,3)
```

In [ ]:
```
f1.add(f2)
f1.print()
```

In [78]:
```
f1.multiply(f2)
f1.print()
```

```
2 / 9
```

In [61]:
```
f.__dict__
```

Out[61]:
```
{'num': 0, 'den': 5}
```

In [62]:
```
f.print()
```

```
0
```

In [63]:
```
f.simplyfy()
```

In [64]:
```
f.print()
```

```
0
```

Complex Numbers

In [128]:
```
class Complex:
    def __init__(self, a = 0, b = 0):
        self.real = a
        self.im = b
        print(self.real, "+ i" , self.im, sep = "")

    def print(self):
        print(self.real, "+ i" , self.im, sep = "")

    def plus(self, other, inplace = False):
        a = self.real + other.real
        b = self.im + other.im
        if inplace == True:
            self.real = a
            self.im = b
            self.print()
        else:
            return (Complex(a,b))


    def mul(self, other):
        a = (self.real* other.real) - (self.im * other.im)
        b = (self.real * other.im) + (self.im * other.real)
        self.real = a
        self.im = b
        self.print()

```

In [129]:
```
c1 = Complex(4,5)
c2 = Complex(6,7)

```

```
4+ i5
6+ i7
```

In [133]:
```
n = c1.plus(c2, inplace = True)
```

```
10+ i12
```

In [134]:
```
c1.print()
```

```
10+ i12
```

In [116]:
```
c1.mul(c2)
```

```
-11+ i58
```

In [121]:
```
n.print()
```

```
10+ i12
```

Inheritance

use **super()** for to inherit attributes from another(parent) class.

In [1]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.maxSpeed = maxSpeed

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def printCar(self):
        print(self.color, self.maxSpeed, self.numGears, self.isConvertible)
```

In [2]:
```
c1 = car('Red', 120, 4, False)
c1.printCar()
```

```
Red 120 4 False
```

Inheritance and Private Members

In [3]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def printCar(self):
        print(self.color, self.__maxSpeed, self.numGears, self.isConvertible)
```

In [4]:
```
c1 = car('Red', 120, 4, False)
c1.printCar()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-8e60d7e7b8f7> in <module>
      1 c1 = car('Red', 120, 4, False)
----> 2 c1.printCar()

<ipython-input-3-48e9c4c1f1c2> in printCar(self)
     11
     12     def printCar(self):
---> 13         print(self.color, self.__maxSpeed, self.numGears, self.isConvertible)

AttributeError: 'car' object has no attribute '_car__maxSpeed'
```

We can not access private variabes from the parent class directly. But we can use functions to return the values of those private variables.

In [5]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def printCar(self):
        print(self.color, self.getMaxSpeed(), self.numGears, self.isConvertible)

    def change_speed()
```

```
  File "<ipython-input-5-5f437f05f6aa>", line 21
    def change_speed()
                      ^
SyntaxError: invalid syntax
```

In [6]:
```
c1 = car('Red', 120, 4, False)
c1.printCar()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-8e60d7e7b8f7> in <module>
      1 c1 = car('Red', 120, 4, False)
----> 2 c1.printCar()

<ipython-input-3-48e9c4c1f1c2> in printCar(self)
     11
     12     def printCar(self):
---> 13         print(self.color, self.__maxSpeed, self.numGears, self.isConvertible)

AttributeError: 'car' object has no attribute '_car__maxSpeed'
```

In [15]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def printCar(self):
        self.Print() # self.print() aslo works fine
        super().Print()
        print(self.numGears, self.isConvertible)

```

self.Print() also works as super().Print()

In [16]:
```
c1 = car('Red', 120, 4, False)
c1.printCar()
```

```
Red 120
Red 120
4 False
```

In [9]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def printCar(self):
        self.Print()  #self.Print() will also work as this function is inherited fr
        print(self.numGears, self.isConvertible)

```

In [10]:
```
c1 = car('Red', 120, 4, False)
c1.printCar()
```

```
Red 120
4 False
```

Methods and properties are inherited from the parent class.

Polymorphism

Method Overwriting

Ability to take multiple forms.

In [20]:

```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def Print(self):
#         self.Print()  #self.Print() will also work as this function is inherited
        print(self.numGears, self.isConvertible)
```

Both Parent(vehicle) and Child(Car) classes have the method **Print** But when called firstly the method from child class is being used.

In [22]:

```
c1 = car('Red', 120, 4, False)
c1.Print()
```

```
4 False
```

In [23]:

```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

#     def Print(self):
#         self.Print()  #self.Print() will also work as this function is inherited
#         print(self.numGears, self.isConvertible)
```

In [24]:
```
c1 = car('Red', 120, 4, False)
c1.Print()
```

```
Red 120
```

First it will search in the child class for the method if not found then it will go for the Parent Class

In [25]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def Print(self):
        self.Print()  #self.Print() will also work as this function is inherited fr
        print(self.numGears, self.isConvertible)

```

In [26]:
```
c1 = car('Red', 120, 4, False)
c1.Print()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-26-75875184239f> in <module>
      1 c1 = car('Red', 120, 4, False)
----> 2 c1.Print()

<ipython-input-25-2ec174f2f597> in Print(self)
     20
     21     def Print(self):
---> 22         self.Print()  #self.Print() will also work as this function is inherit
ed from the parent class
     23         print(self.numGears, self.isConvertible)
     24

... last 1 frames repeated, from the frame below ...

<ipython-input-25-2ec174f2f597> in Print(self)
     20
     21     def Print(self):
---> 22         self.Print()  #self.Print() will also work as this function is inherit
ed from the parent class
     23         print(self.numGears, self.isConvertible)
     24

RecursionError: maximum recursion depth exceeded
```

Max recursion error because the Print function calls itself again and again because it finds the

function *Print()* in itself

To avoid from this we need ***Super()***. *Super()* ensures that function is being called from Parent class.

In [28]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self.__maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self.__maxSpeed

    def setMaxSpeed(self, newSpeed):
        self.__maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self.__maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def Print(self):
        super().Print()  #self.Print() will also work as this function is inherited
        print(self.numGears, self.isConvertible)

```

In [29]:
```
c1 = car('Red', 120, 4, False)
c1.Print()
```

```
Red 120
4 False
```

Protected Members

Protected members are just public members but a underscore(_) indicates that it is inherited from a parent class we should not access or modify this.

In [33]:
```
class vehicle:
    def __init__(self, color, maxSpeed):
        self.color = color
        self._maxSpeed = maxSpeed

    def getMaxSpeed(self):
        return self._maxSpeed

    def setMaxSpeed(self, newSpeed):
        self._maxSpeed =  newSpeed

    def Print(self):
        print(self.color, self._maxSpeed)

class car(vehicle):
    def __init__(self, color, maxSpeed, numGears, isConvertible):
        super().__init__(color, maxSpeed)
        self.numGears = numGears
        self.isConvertible = isConvertible

    def Print(self):
        super().Print()  #self.Print() will also work as this function is inherited
        print(self.numGears, self.isConvertible)

```

In [34]:
```
c1 = car('Red', 120, 4, False)

```

In [35]:
```
c1._maxSpeed
```

Out[35]:
```
120
```

Class "object"

Every class inherit from **object** class.

object class have 3 methods.

1. --new--
2. --init--
3. --str--

In [36]:
```
class Circle:
    def _init__(self,radius):
        self.radius = radius
```

In [37]:
```
c = Circle()
print(c)
```

```
<__main__.Circle object at 0x00000214500627F0>
```

--str-- function prints the location of the object. But we can override this function to print some useful information about the class.

In [38]:
```
class Circle:
    def _init__(self,radius):
        self.radius = radius

    def __str__(self):
        return "This is a class Cicle which takes radius as an argument"
```

In [40]:
```
c = Circle()
print(c)  # overriding the __str__() function and print the custom information.
```

```
This is a class Cicle which takes radius as an argument
```

Multiple Inheritance

In [41]:
```
class mother:
    def print(self):
        print("print of mother called")

class father:
    def print(self):
        print("print of father called")

class child(mother, father):
    def __init__(self,name):
        self.name = name
    def printChild(self):
        print("name is", self.name)
```

In [42]:
```
a = child("Rohan")
a.printChild()
```

```
name is Rohan
```

In [43]:
```
a.print()
```

```
print of mother called
```

print of mother is called as we have written child(mother, father) is mother first.

as we wrote father first the print of father is called

In [52]:
```
class mother:
    def __init__(self):
        self.name = "Manju"


    def print(self):
        print("print of mother called")

class father:
    def __init__(self):
        self.name = "Ajay"

    def print(self):
        print("print of father called")

class child(father, mother):
    def __init__(self):
        super().__init__()


    def printChild(self):
        print("name is", self.name)
```

In [54]:
```
a = child()
a.printChild()
```

```
name is Ajay
```

Method Resolution Order

This tells about the hirearcy order of the classes properties and mehods for multiple parent classes.

In [2]:
```
class mother:
    def __init__(self):
        self.name = "Manju"


    def print(self):
        print("print of mother called")

class father:
    def __init__(self):
        self.name = "Ajay"

    def print(self):
        print("print of father called")

class child(father, mother):
    def __init__(self):
        super().__init__()


    def print(self):
        print("name is", self.name)
```

In [3]:
```
c = child()
c.print()
```

```
name is Ajay
```

In [5]:
```
# print the method resolution order
print(child.mro())
```

```
[<class '__main__.child'>, <class '__main__.father'>, <class '__main__.mother'>, <clas
s 'object'>]
```

mro is child > father > mother > object class

In [9]:
```
class father:
    def __init__(self):
        self.name = "Ajay"
        super().__init__()

    def print(self):
        print("print of father called")


class mother:
    def __init__(self):
        self.name = "Manju"



    def print(self):
        print("print of mother called")

class child(father, mother):
    def __init__(self):
        super().__init__()


    def print(self):
        print("name is", self.name)
```

In [10]:
```
c = child()
c.print()
```

```
name is Manju
```

using super() we go to one superior class to the present class

In this case as father was super class of child and mother is super class of father we used init() in the father class that changes the order to mother which is its super class.

father is super class of child and mother is super class of father.

Operator Overloading

In [28]:
```
import math
class Point:
    def __init__(self,x,y):
        self.__x = x
        self.__y = y
    def __str__(self):
        return "This point is at " + str(self.__x) + "," + str(self.__y)

    def __add__(self, point_obj): # overloading + operator
        return Point(self.__x + point_obj.__x , self.__y + point_obj.__y)

    def __lt__(self, po): #overloading < operator
        return math.sqrt(self.__x**2 + self.__y**2) < math.sqrt(po.__x**2 + po.__y*

```

In [29]:
```
p1 = Point(2,3)
p2 = Point(4,5)
print(p1+p2)  # + operator overloading
```

```
This point is at 6,8
```

In [31]:
```
print(p2<p1)
```

```
False
```

Abstract Classes

Abstract Classes contains abstract methods

Those methods which are declared but not implemented. They will be implemented by some other class.

In [14]:
```
from abc import ABC, abstractmethod
class automobile(ABC):

    def __init__(self):
        print("automobile created")


    def start(self):
        pass


    def drive(self):
        pass


    def stop(self):
        pass
```

In [15]:
```
c = automobile()
```

```
automobile created
```

In [16]:
```
from abc import ABC, abstractmethod
class automobile(ABC):

    def __init__(self):
        print("automobile created")

    @abstractmethod
    def start(self):
        pass

    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass
```

In [17]:
```
c = automobile()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-33fbd8d848ba> in <module>
----> 1 c = automobile()

TypeError: Can't instantiate abstract class automobile with abstract methods drive, st
art, stop
```

can't create object of the class which have any abstract methods

In [18]:
```
class Car(automobile):

    def __init__(self,name):
        print("car created")
        self.name = name

```

In [19]:
```
a = Car("Honda")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-5935755da3ab> in <module>
----> 1 a = Car("Honda")

TypeError: Can't instantiate abstract class Car with abstract methods drive, start, st
op
```

We can't initate an object of Car without using the abstract methods of automobile class

In [21]:
```
class Car(automobile):

    def __init__(self,name):
        print("car created")
        self.name = name

    def start(self):
        pass

    def stop(self):
        pass

    def drive(self):
        pass

```

In [22]:
```
c = Car("Honda")
```

```
car created
```

Important Points about Abstract Methods

1. Objects of abstract class can not be created.
2. Implement all the abstract methods in the child class.

In [28]:
```
from abc import ABC, abstractmethod
class automobile(ABC):

    def __init__(self):
        print("automobile created")

    @abstractmethod
    def start(self):
        print("start of automobile called")
        pass

    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass

class Car(automobile):

    def __init__(self,name):
        print("car created")
        self.name = name

    def start(self):
        super().start()
        print("start of car called")
        pass

    def stop(self):
        pass

    def drive(self):
        pass
```

In [29]:
```
c = Car("Maruti")
```

```
car created
```

In [30]:
```
c.start()
```

```
start of automobile called
start of car called
```

In [33]:
```
from abc import ABC, abstractmethod
class automobile(ABC):

    def __init__(self, no_of_wheels):
        self.wheels = no_of_wheels
        print("automobile created")

    @abstractmethod
    def start(self):
        print("start of automobile called")
        pass

    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass

class Car(automobile):

#     def __init__(self,name):
#         print("car created")
#         self.name = name

    def start(self):
        super().start()
        print("start of car called")
        pass

    def stop(self):
        pass

    def drive(self):
        pass
```

In [34]:
```
b = Car(4)
```

```
automobile created
```